Programmation Orientée Objets 2
Cours 7

Rémi Watrigant[1]

[1] basé sur le cours de Marianne Huchard

Sommaire du cours 7

Gestion des exceptions en C++

1. Modélisation
2. Définition d'exceptions en C++
3. Déclaration/signalement
4. Récupération
5. API
6. Exceptions du mécanisme

Exemple

```
int division(int a, int b) {
   return a/b;
}

int main() {
    int a, b;
    cout << "valeur pour a : " ;
    cin >> a;
    cout << "valeur pour b : ";
    cin >> b;

    cout << a << "/" << b << "=" << division(a, b) << endl;

    return 0;

 }
```

Exemple

```
int division(int a, int b) {
   return a/b;
}

int main() {
    int a, b;
    cout << "valeur pour a : " ;
    cin >> a;
    cout << "valeur pour b : ";
    cin >> b;

    cout << a << "/" << b << "=" << division(a, b) << endl;

    return 0;

 }
```

Exemple d'exécution :
Valeur pour a : 3
Valeur pour b : 0
Exception en point flottant (core dumped)

Exemple : trois solutions

```
int ERROR_VALUE = ..... ?

int division(int a, int b) {
    if (b != 0)
        return a/b;
    else
        return ERROR_VALUE;
}
```

```
int division(int a, int b) {
    if (b != 0) {
        return a/b;
    else
        cerr << "erreur : division par zéro" << endl;
}

bool division(int a, int b, int& resultat) {
    if (b != 0) {
        return resultat = a/b;
        return true;
    } else {
        return false;
    }
}
```

Motivation

Séparer le code *normal* du code de *gestion des situations exceptionnelles*

- Clarification du code source
- Résistance des programmes aux erreurs
- Meilleure réutilisation des opérations

Résistance des programmes aux erreurs

- détecter certaines erreurs
- prendre en charge une partie de leur traitement
- en cas d'erreur, pas d'arrêt brutal du programme
- déroutement de son fil normal
- assurer la cohérence des données après exécution
  - ⇒ procédures de sauvegarde des données, de mise à jour ou de retour vers des versions antérieures

Meilleure réutilisation des opérations

- extension du domaine de définition
- pré-conditions vérifiées
- violations des pré-conditions prises en charge par le mécanisme de gestion d'exceptions

Contrat d'une opération

Pré-conditions

- assertions qui doivent être vraies avant d'entrer dans le code de l'opération, pour interdire :
  - Diviser par `zéro`
  - `depiler` une pile vide
  - `setJour` d'une `Date` avec la valeur 32

Contrat d'une opération

Post-conditions

- assertions qui doivent être vraies après le retour de l'opération pour éviter :
  - Débordement dans une opération arithmétique
  - écriture d'un fichier sur une disquette pleine
  - réservation d'un emplacement dans une mémoire saturée

Contrat d'une classe

Invariants

- assertions vérifiées par les objets d'une classe, très souvent des propriétés sur les valeurs des attributs
  - Dans une classe `Personne`, l'attribut age doit varier entre 0 et 140
  - Si on stocke dans un attribut la date de naissance et dans un autre attribut l'âge, ils doivent toujours être cohérents

Sans mécanisme de gestion d'exceptions

- *Traiter le problème dans l'algorithme de l'opération*
  - → le code de l'opération est alourdi
  - → le traitement des erreurs est figé dans l'opération
- *L'opération retourne un code d'erreur*
  - → le code appelant l'opération doit se préoccuper du traitement des situations exceptionnelles, et possède le même défaut de lourdeur
  - → Le traitement est cependant moins figé car il est traité plus près du contexte où l'erreur s'est produite
  - → Le code d'erreur peut être plus ou moins précis (entier, chaîne de caractères, objet)

Exceptions dans les langages de programmation

- dans la plupart des langages modernes
- dans le cadre des langages à objets, les erreurs sont le plus souvent représentées par des objets

Utilisation en C++

- block try { .... } où une erreur peut survenir (dans le code ou les fonctions appelées)
- mot clé throw pour lancer une exception lors d'une erreur
- zone catch(..) { ... } pour attraper l'exception lancée et gérer l'erreur.

Utilisation en C++

```
//ici du code sûr
bool lancerException = true;

try {
    //ici du code succeptible de lancer une erreur
    if (lancerException) {
         string s = "je lance une erreur!!";
         throw s;

         //cette zone ne sera jamais exécutée
    }
    //cette zone sera exécutée seulement si lancerException == false;

} catch (string & e) {
    //ici on attrape une exception de type string
   cout << "erreur rattrapée! : " << endl;
   cerr << e << endl;

} catch (int e) {
   //ici on attrape une exception de type int
   cerr << "erreur rattrapée : " << e << endl;
}

//le code reprend ici !
```

Dans notre exemple de division

1ère solution :

```
int division(int a, int b) {
    try {
        if (b != 0)
            return a/b;
        else
            throw string("division par zéro!");
    } catch (string& e) {
        cerr << e << endl;
    }
}
```

pas très différent des "mauvaises" solutions...!
On voudrait une fonction $division(int, int)$ sans effet de bord

```
int division(int a,int b) {
   if (b != 0)
       return a/b;
   else
      throw string("division par zéro!");
}

int main()
{
    int a,b;
    cout << "Valeur pour a : ";
    cin >> a;
    cout << "Valeur pour b : ";
    cin >> b;

    try{
        cout << a << " / "<< b << " = " << division(a,b) << endl;
    } catch(string chaine) {
        cerr << chaine << endl;
    }
    return 0;
}
```

Les erreurs sont des objets

- La récupération des exceptions se fait d'après des types (leurs classes), leur sémantique est donc explicite et facilite la lisibilité du programme
- Les types d'exceptions peuvent être classés grâce à l'héritage
- Les objets exceptions peuvent transporter des informations (dans leurs attributs) depuis le contexte où l'erreur est survenue jusqu'au contexte où elle est traitée
- certains traitements typiques peuvent être intégrés dans les opérations de la classe exception
- Par exemple un programme qui effectue des traitements sur des fichiers rencontre une erreur (fichier de mauvais type, corrompu...etc) peut lancer une exception CorruptedFileException qui contient le chemin du fichier qui pose problème.

Les erreurs sont des objets

La STL fournit des classes d'exceptions générales

```
class exception
{
public:
    exception() throw(){ }
    virtual ~exception() throw();

    virtual const char* what() const throw();  // Renvoie une chaîne contenant des i
};
```

Pour un code portable : dériver de la classe *Exception*

```
#include <exception>
using namespace std;
```

Les exceptions de la STL

Plusieurs types d'exception déjà définis :

- bad_alloc Lancée s'il se produit une erreur lors d'une manipulation de la mémoire.
- bad_cast Lancée s'il se produit une erreur lors d'un dynamic_cast.
- bad_exception Lancée si aucun catch ne correspond à un objet lancé.
- bad_typeid Lancée s'il se produit une erreur lors d'un typeid.
- ios_base::failure Lancée s'il se produit une erreur avec un flux.
- domain_error A Lancer s'il se produit une erreur de domaine mathématique.
- invalid_argument A Lancer si un des arguments d'une fonction est invalide.
- length_error A Lancer si un objet aura une taille invalide. Par exemple si la classe Pile vue précédemment a une taille dépassant la taille de la mémoire.
- out_of_range A Lancer s'il y a une erreur avec un indice. Par exemple si on essaye d'accéder à une case inexistante d'un tableau.
- logic_error A Lancer lors de n'importe quel autre problème de logique du programme.
- range_error A Lancer lors d'une erreur de domaine à l'exécution.
- overflow_error A Lancer s'il y a une erreur d'overflow.
- underflow_error A Lancer s'il y a une erreur d'underflow.
- runtime_error A Lancer pour tout autre type d'erreur non-prévue survenant à l'exécution.

Les exceptions de l'API C++

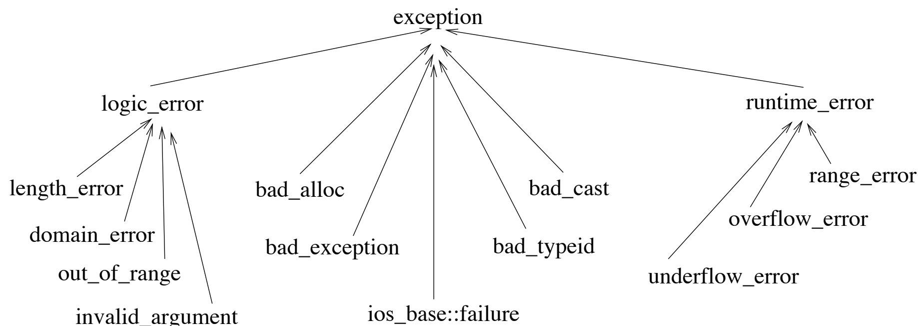


Figure: Complete hierarchy of C++ exceptions

Le programmeur est invité à ajouter ses propres erreurs dans la hiérarchie, mais il n'y a pas d'obligation

Les exceptions de la STL

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    try{
        vector<int> a(1000000000,1);

    }  catch(exception const& e)  {
        cerr << "ERREUR : "<< e.what() << endl;
    }
    return 0;
}
```

Cas d'étude

```
//--------------- Date .h----------------------------------------------------
class Date
{
private:
    int annee, mois, jour;
public:
    Date();
    Date(int,int,int);
    virtual ~Date();
    virtual int getAnnee()const;
    virtual int getMois()const;
    virtual int getJour()const;
    virtual void setAnnee(int);
    virtual void setMois(int);
    virtual void setJour(int);
    virtual void modifier(int j, int m, int a);
    virtual void affiche(ostream&)const;
    static int NbrJoursMois(int, int);
};
```

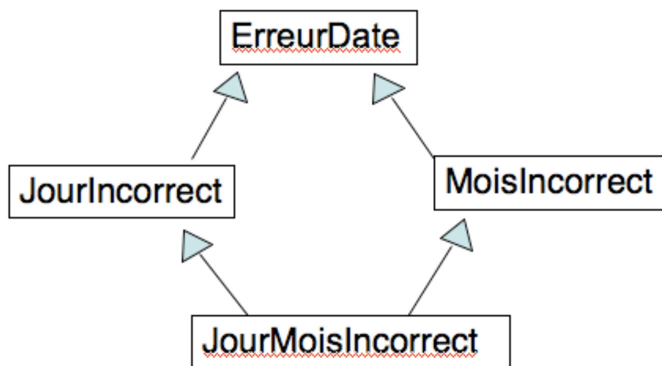
Cas d'étude

```
//-------------- Date .cpp---------------------------------------------

Date::Date(int j, int m, int a) {
    jour = j; mois = m; annee = a;
}

int Date::getAnnee()const{
    return annee;
}

...

void Date::setMois(int m){
    mois=m;
}
...
```

Erreurs étudiées de la classe `Date`

- une valeur de jour erronée
- une valeur de mois erronée
- la combinaison des deux erreurs précédentes

Erreurs étudiées de la classe Date



ErreurDate

```
class ErreurDate : public Exception {
protected:
    int jour;
    int mois;
    int annee;
public:
   ErreurDate();
   ErreurDate(int, int, int);
   virtual ~ErreurDate();
   virtual char* what();
};

ErreurDate::ErreurDate(){
}

ErreurDate::ErreurDate(int j, int m, int a) : jour(j), mois(m), annee(a){
}

ErreurDate::~ErreurDate(){
}

char* ErreurDate::what(){
    string returnString = "mauvaise date : " + jour + "/" + mois + "/" + annee ;
    return returnString.c_str();
}
```

JourIncorrect

```
class JourIncorrect : virtual public ErreurDate {
public:
   JourIncorrect();
   JourIncorrect(int, int, int);
   virtual ~JourIncorrect();
   virtual char* what() ;
};

char* JourIncorrect::what() {
string returnString;
    switch (mois) {
         case 4:case 6:case 9:case 11:
                  returnString = "30 jours pour le mois numero "  + mois ;
                  break;
         case 2: if ((Date::NbrJoursMois(mois, annee)==28))
                 returnString = "28 j. en fevrier les annees non bissextiles !" ;
                 break;
        default:
                returnString = "31 jours pour le mois numero " ;
    }
    return returnString.c_str();
}
```

MoisIncorrect

```
class MoisIncorrect : virtual public ErreurDate {
    public:
          MoisIncorrect(int, int, int);
          virtual ~MoisIncorrect();
          virtual char* what();
};

char* MoisIncorrect::what()  {
     string returnString = mois + " est un numero de mois incorrect" ;
     return returnString.c_str();
}
```

JourMoisIncorrect

```
class JourMoisIncorrect : virtual public JourIncorrect, virtual public MoisIncorrect
public:
    JourMoisIncorrect(int,int,int);
    virtual ~JourMoisIncorrect();
    virtual char* what();
};

char* JourMoisIncorrect::what()  {
    string returnString = "Jour et mois sont incorrects" ;
    return returnString.c_str();
}
```

Signalement des exceptions

Le signalement d'une exception se réalise grâce à l'instruction `throw`

```
void  Date::modifier(int j, int m, int a) {

  if (((m<1) || (m>12)) && ((j<1) || (j>NbrJoursMois( m, a))))
      throw JourMoisIncorrect(j,m,a);

  if ((m<1) || (m>12))
      throw MoisIncorrect(j,m,a);

  if ((j<1) || (j>NbrJoursMois( m, a)))
      throw JourIncorrect(j,m,a);

  jour = j; mois = m; annee = a;
}
```

Déclaration des exceptions

- dans la signature
- pour une méthode de `Date` dans le fichier Date.h et dans le fichier Date.cc
- la méthode ne peut alors signaler que des exceptions qui sont des objets des classes mentionnées (ou de leurs sous-classes)

Déclaration des exceptions

lorsque l'on ne met rien, n'importe quelle exception est susceptible d'être signalée par la méthode :

```
void modifier(int j, int m, int a)
```

Lorsque l'on met une clause de spécification d'exception, cela limite les exceptions signalables par la méthode.

```
void modifier(int j, int m, int a)
                throw (ErreurDate,ErreurCalendrier)
```

Déclaration des exceptions

- redéfinition de la méthode dans une sous-classe : restriction des exceptions déclarées
  - retrait
  - spécialisation

Dans une sous-classe de `Date` on pourra ainsi redéfinir `modifier` :

```
void modifier(int j, int m, int a) throw (JourIncorrect)
```

Mais on ne pourra pas ajouter une nouvelle exception
⇒ qui ne serait pas sous-classe de `ErreurDate` et `ErreurCalendrier`

try ... catch

`try` *BlocProtégé*
`catch` *(déclaration1) BlocTraitement 1*
`catch` *(déclaration2) BlocTraitement 2*
...

Schématisation du mécanisme

▷ `throw(e)` $\rightarrow$ interruption du programme
▷ Propagation sur les blocs `try .. catch ..` englobants jusqu'au premier `catch` qui contient une déclaration compatible avec `e` et destruction au passage de tous les objets locaux créés dans le bloc `try` ou les fonctions qu'il appelle
▷ Exécution du bloc du `catch` avec liaison de `e` avec le paramètre formel du `catch`
▷ Continuation par défaut : instruction suivant l'instruction `try`

Conformité de la date de saisie

```
int main()
{
 Date D;
int i, j, m, a;
bool DateCorrecte = false;

while (!DateCorrecte)
{
   cout << "date ? "; cin >> j >> m >> a;
   try {
       D.modifier(j, m, a);
       DateCorrecte = true;
   }
   catch (JourMoisIncorrect JMI) {
       cerr << JMI.what() << endl;
   }
   catch (JourIncorrect JI) {
       cerr << JI.what() << endl;
   }
   catch (MoisIncorrect MI) {
       cerr << MI.what() << endl;
  }
}
}
```

Imbrication de blocs de récupération

```
try
{
  try
  {
    try {... /* levée de Z() */ ...}
    catch(X) {...}
    catch(Y) {...}
    .....
  }
  catch(Z) {...}
  catch(V) {...}
  ..... /* reprise ici */
}
catch(U) {...}
....
```

Problèmes du mécanisme de gestion d'exceptions

Gérés par appel d'une fonction que l'on peut modifier

▷ une exception est signalée qui n'est prévue dans aucun `try..catch..`

la fonction `terminate()` *qui appelle* `abort` *par défaut est appelée*

▷ une exception est signalée qui n'est pas prévue dans la déclaration de la fonction

la fonction `unexpected()`*, qui appelle* `terminate()` *par défaut, est appelée*

Dans les deux cas, terminaison anormale du programme, mais on peut redéfinir la terminaison, en utilisant les fonctions `set_terminate` et `set_unexpected`

Problèmes du mécanisme de gestion d'exceptions

```
// ------------------ déclaration de deux fonctions ---------------
void f(){cout << "le programme va terminer brutalement" << endl;}
void fu(){cout << "une erreur due a un probleme de specification
     d'exception" << endl;}

// ------ association de ces fonctions aux fonctions de terminaison
// ------ anormales
set_terminate(f);
set_unexpected(fu);
```

Problèmes du mécanisme de gestion d'exceptions

```
//------------------ cas d'appel de terminate

int main() {Date D; D.modifier(34, 64, 1997);}
```

Problèmes du mécanisme de gestion d'exceptions

```
//------------------ cas d'appel de unexpected


class ErreurJustePourIllustrerUnexpected{};

void  Date::modifier(int j, int m, int a) throw (ErreurDate){
throw ErreurJustePourIllustrerUnexpected();
......... }



int main() {Date D; D.modifier(34, 64, 1997);}
```